MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

3-tier REST Services

Hands on SPRING REST Framework

Ioan Salomie
Marcel Antal

Tudor Cioara
Claudia Daniela Pop

Ionut Anghel
Cristina Pop

Contents

1. Introduction

The Spring Framework is an application framework for the Java platform. The framework was first released on the 1st October 2002 and was written by the Australian computer specialist Rod B. Johnson. Due to its continuous enhancement and development, the framework became widely used in software companies nowadays. The project example from this laboratory work is a skeleton for a Spring application that can be used to get the information associated with the users described in a database.

1.1. Get the project

1. Setup GIT and download the project from https://gitlab.com/ds2025/ds2025_spring_example/-/tree/main
   - Create an empty local folder in the workspace on your computer
   - Right-click in the folder and select **Git Bash**
   - Write the commands:
     - git clone https://gitlab.com/ds2025/ds2025_spring_example/-/tree/main
2. Create an empty database in PostgreSQL with the name **example-db**
3. Import the project in IntelliJ
4. Check the **application.properties** file from src/main/resources and fill the database.user and database.password of the local PostgreSQL server.
5. In the **application.properties** file change the ddl-auto flag from validate to create in order for the Spring Application to be able to create the structure of the tables:
   - *spring.jpa.hibernate.ddl-auto = create*

   Observation! Make sure you change the property back to **validate** or **update**, in order to avoid recreating the database at the following restart.
6. Run the Application in IntelliJ: Right-click on the class DemoApplication and select **Run 'DemoApplication'**. (Figure 1)
7. (Optional) Go to your workbench application (pgAdmin for PostgreSQL or MySQL Workbench) and insert a person in the database using the following query

```
INSERT INTO person (id, name, address, age)
VALUES (decode(replace('45774962-e6f7-41f6-b940-72ef63fa1943'::text,
'-', ''), 'hex'), 'Its me', 'My Address', 22);
```

Observation! Once inserted the Person in the database, the UUID will be stored in a binary format, thus the id retrieved should not be copied in the plain form, but should be hex encoded.

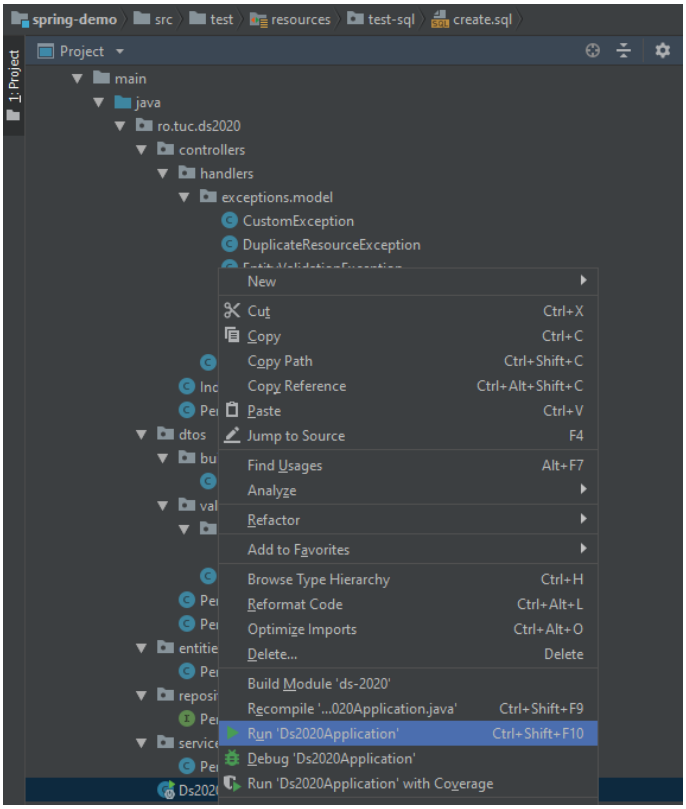

Figure 1 Run Application

8. Test the implemented REST requests:

- http://localhost:8080/people => this request should retrieve all the persons from the database
- http://localhost:8080/people/45774962-e6f7-41f6-b940-72ef63fa1943 => this request should retrieve the person with the UUID 45774962-e6f7-41f6-b940-72ef63fa1943


```
{
    "id": "45774962-e6f7-41f6-b940-72ef63fa1943",
    "name": "Its me",
    "age": 22
}
```


Figure 2 Result of the Query that Retrieves a Person by UUID from the Database

9. The project can also be tested from IntelliJ, running the instructions clean and install. Upon successful building, 8 out of the 8 tests should run successfully.

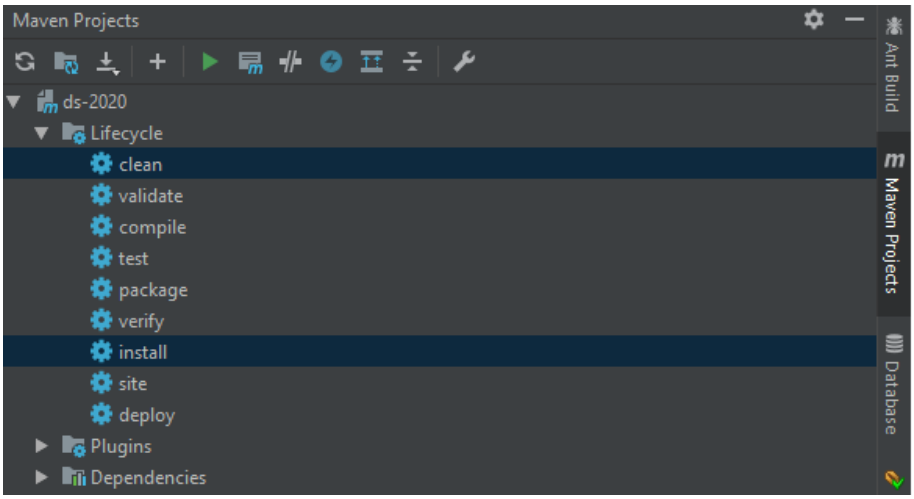

Figure 2 Build the Project in IntelliJ

```
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.3:report (generate-code-coverage-report) @ ds-2020 ---
[INFO] Loading execution data file E:\DS2020\spring-demo\target\jacoco.exec
[INFO] Analyzed bundle 'ds-2020' with 16 classes
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.3:report (post-unit-test) @ ds-2020 ---
[INFO] Loading execution data file E:\DS2020\spring-demo\target\jacoco.exec
[INFO] Analyzed bundle 'ds-2020' with 16 classes
[INFO]
```

Figure 3 Test Results

The provided project is a Spring Boot JAR application. You can start it directly using **mvn spring-boot:run** or by running DemoApplication in IntelliJ. No external Tomcat deployment is required.

1.2. Project Structure

The project presents some basic operations on the person entity and aims at presenting the layers involved in performing CRUD operations on the person table shown in the following figure:

When opened in IntelliJ, the project has the following structure shown in Figure 7. All the components are detailed in Chapter 2.

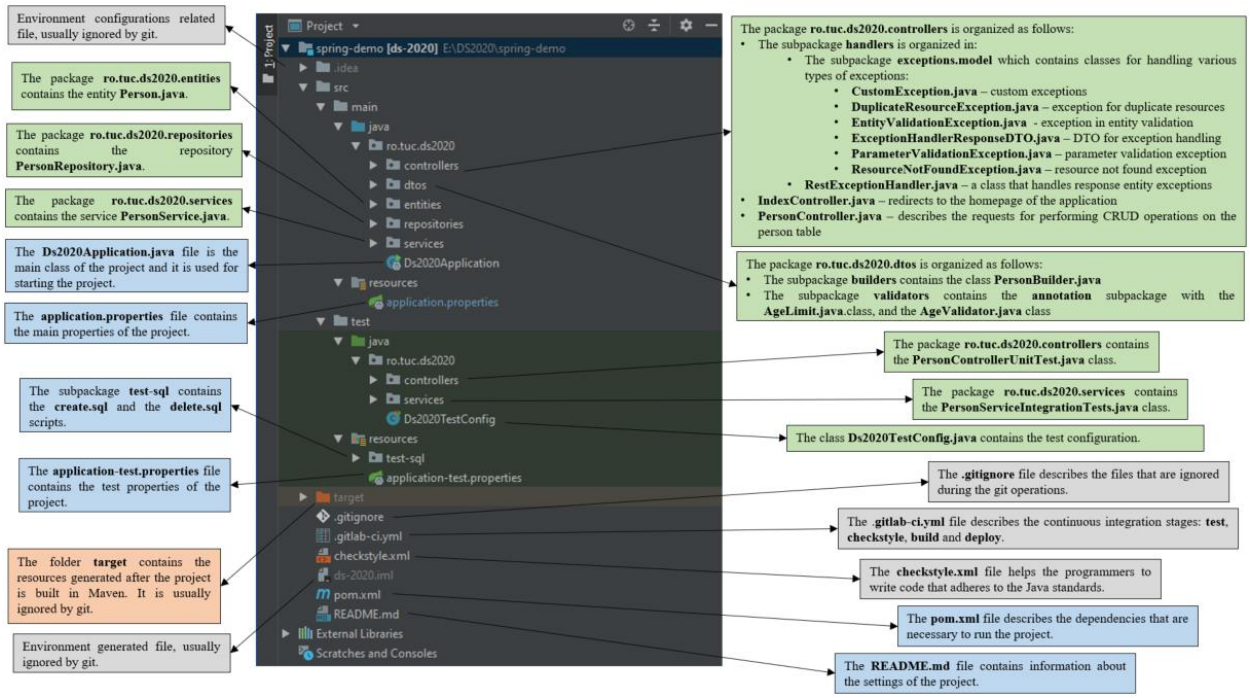

Figure 4 Project Structure in IntelliJ

2. Project Conceptual Architecture

The conceptual architecture of the system is presented below.

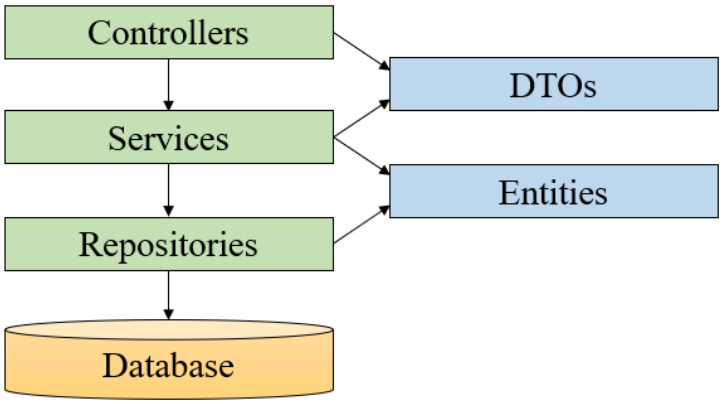

Figure 5 Project Conceptual Architecture

The following table describes each component:

Table 2-1- Project Components Description

Component	Package	Description
Repositories	com.example.demo.repositories	Package that contains the repositories, classes that facilitate the DB access. The developer can use custom queries to communicate with the DB.
Entities	com.example.demo.entities	An entity corresponds to a table from the relational database and each instance of the entity corresponds to a row from the database
Services	com.example.demo.services	This layer represents the business logic layer of the Spring application. It translates the Data Transfer Objects (DTOs) into entities and back. For formatting the values from DTO objects to Entity objects Builder classes are used. The service layer is responsible to apply more complex operations and validations before accessing the repository layer.
DTOs	com.example.demo.dtos	A Data Transfer Object (DTO) is a special object exposed outside the application (to the UI or APIs). It contains part of the underlying Entities or combinations of different entities. Additionally, it contains builders and validators.
Controller	com.example.demo.controllers	The layer that exposes the application functionality as an API able to handle HTTP REST requests. It also contains handlers for various types of exceptions.

3. Project Functionality

A simple sequence diagram that involves the interactions between the components is shown in Figure 7:

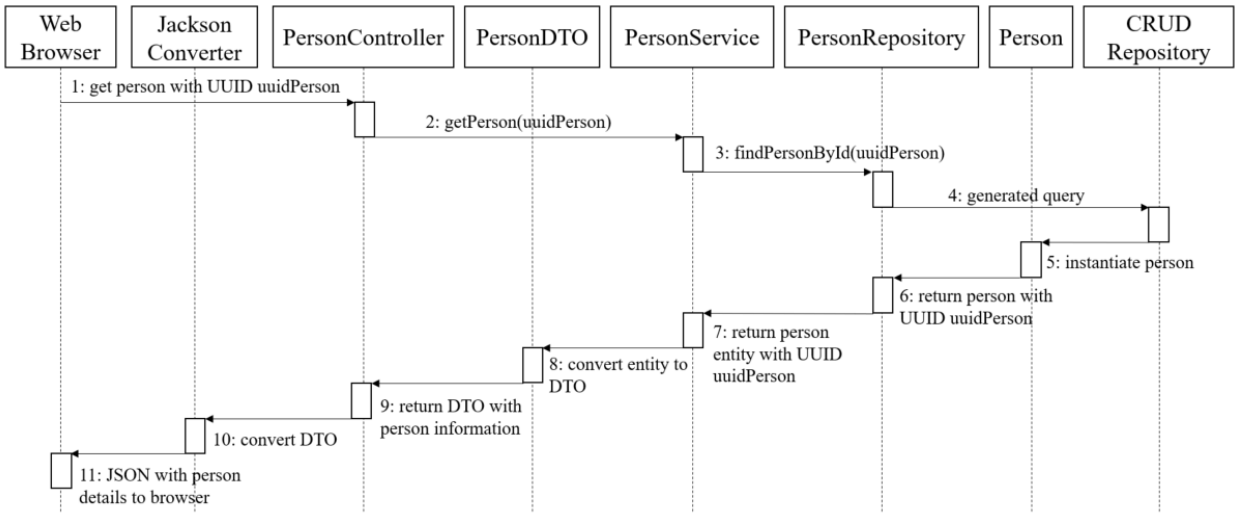

Figure 6 Sequence Diagram for GET operation

The processing steps for a person with the uuidPerson 45774962-e6f7-41f6-b940-72ef63fa1943 are described in the following section:

1. Web Browser sends a HTTP request with the method GET to retrieve the user with UUID = 45774962-e6f7-41f6-b940-72ef63fa1943. This happens by calling the URL: http://localhost:8080/people/45774962-e6f7-41f6-b940-72ef63fa1943. This URL is composed of the following parts:
   - **http**: protocol used to communicate
   - **localhost**: address of the server to communicate with. This can be either an URL resolved by DNS to an IP address, or an IP address. (localhost or 127.0.0.1 in this case).
   - **8080**: the port on which the web server which will respond to the request is listening.
   - **http://localhost:8080/people/45774962-e6f7-41f6-b940-72ef63fa1943 .** The last part of the address is mapped to the resources within the application by the web server. In this case, the application exposes a REST API through its **controllers**. The mapping is done in com.example.demo.controllers.PersonController.java in three steps, as follows:
     i) mapping to the controller: @RequestMapping(value = "/people") (line 18)
     ii) mapping to the method within the controller and defining the request type: @GetMapping(value = "/{id}") (line 40)
     iii) defining the parameters of the method at (line 40) and (line 41) (the name in the request must correspond to the name within the

@PathVariable tag. Inside the method, the Java Parameter is used – **int id**).

2. The Spring controller com.example.demo.controllers.PersonController.java has already instantiated a service instance due to the annotation @Autowired (line 23). Using this com.example.demo.services.PersonService.java object, inside the getPerson() method it calls the findPersonById() method (line 42), delegating the processing to the service layer.

Good to know

The **Controllers Layer** is a layer over the **Services Layer** and calls the methods which are provided by the **Services Layer**.

a) How are the controllers defined?

The controllers are defined using the annotation @RestController. This annotation specifies the fact that the corresponding annotated class can handle RESTful WEB Services. The REST (Representational State Transfer) services describe one way of communication between different computer systems on the internet. REST uses HTTP (Hyper Text Transfer Protocol) for the communication with the internet resources.

b) How are the controllers mapped to the URLs?

The controllers are mapped to the URLs using mapping annotations. These annotations are used in two cases:

- For annotating the entire class – in this case the value of the @RequestMapping (line 17) is a **prefix** for all the other URLs that are handled by the controller.

```
16  @RestController  gobia
17  @RequestMapping("/people")
18  @Validated
19  public class PersonController {
20
21      private final PersonService personService;  4 usages
```

Figure 7 RequestMapping for PersonController

- For annotating a specific method – in this case the value of the @GetMapping (line 40) is a **suffix** for the URL that corresponds to the method. The parameter **value** describes the location while the request method is inferred from the name of the annotation. There are various types of mapping annotations such as: @GetMapping, @PostMapping, @PutMapping and @DeleteMapping.

```
27      @GetMapping  gobia
28      public ResponseEntity<List<PersonDTO>> getPeople() {
29          return ResponseEntity.ok(personService.findPersons());
30      }
```

Figure 8 GetMapping for getting all the persons

c) How are the **Services Layer** instances accessed?

- The objects from the **Services Layer** are accessed using the @Autowired annotation.

d) What are the most common input parameters of the methods annotated with @GetMapping or @PostMapping?

- A path variable – in this case the variable is a part of the path specified by the @GetMapping annotation

```
43      @GetMapping("/{id}")  gobia
44      public ResponseEntity<PersonDetailsDTO> getPerson(@PathVariable UUID id) {
45          return ResponseEntity.ok(personService.findPersonById(id));
46      }
```

Figure 9 Example that uses the @PathVariable annotation

- A request body – in this case the object is a DTO that contains information to be inserted in the application

```
32      @PostMapping  gobia *
33      public ResponseEntity<Void> create(@Valid @RequestBody PersonDetailsDTO person) {
34          UUID id = personService.insert(person);
```

Figure 10 Example that uses the @RequestBody annotation

3. The controller calls personService.findPersonById(id) using **constructor-based injection**. The service retrieves the person from the repository by UUID and returns a PersonDetailsDTO to the controller, which wraps it in a ResponseEntity.

Good to know

The **Services Layer** is an intermediary layer between the **Repositories Layer** and the **Controllers Layer**.

a) What are the services?

The services provide transactional operations for the business logic. A service method either completes or the database rollbacks to the previous state.

b) What is the purpose of the **Services Layer**?

The purpose of the **Services Layer** is to define methods that perform several operations on a database in such a way that either all the operations execute successfully or none of them is executed. In the second case the database rollbacks to the original state. It is responsible to process the data before accessing the repository layer, in order to ensure that all the business logic rules hold true.

c) How are the services defined?

The services are defined using the annotation @Component or the annotation @Service.

d) How are the objects defined in the **Repositories Layer** accessed?

The repository is injected through the service's **constructor**, optionally marked with @Autowired, following the recommended **constructor-based dependency injection** pattern.

```java
@Service  4 usages  gobia
public class PersonService {
    private static final Logger LOGGER = LoggerFactory.getLogger(PersonService.class);  2 usages
    private final PersonRepository personRepository;  4 usages

    @Autowired  gobia
    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }
```

Figure 11 PersonService

e) Why does the Services Layer use DTOs instead of entities?

Usually, the DTOs reduce the overhead between the backend and the presentation. The optimized DTOs contain only that information which is absolutely required. Furthermore, the purpose of the DTO is also to restrict the access to the information exposed by the entities. Depending on the roles/ rights of the requesting clients, some information may not be allowed to be exposed, thus DTOs can be used to restructure the information exposed on the external services.

4. he PersonRepository method findById is called, using an auto-generated query from JpaRepository<Person, UUID> to retrieve a person by its UUID. The repository also defines additional methods such as findByName (derived query) and findSeniorsByName (custom query using @Query and @Param annotations).

Good to know

The **Repositories Layer** intermediates the communication between the **Services Layer** and the **Database**.

a) How are the repositories defined?

The repositories are defined by extending the interface **JpaRepository<T, ID extends Serializable>**. The first argument **T** describes the type of the entities used by the repositories while the second argument **ID** describes the type of the id of the entities.

```java
public interface PersonRepository extends JpaRepository<Person, UUID> {

    /**
     * Example: JPA generate Query by Field
     */
    List<Person> findByName(String name);

    /**
     * Example: Write Custom Query
     */
    @Query(value = "SELECT p " +
            "FROM Person p " +
            "WHERE p.name = :name " +
            "AND p.age >= 60 ")
    Optional<Person> findSeniorsByName(@Param("name") String name);

}
```

Figure 11 PersonRepository interface

The **PersonRepository** handles entities of the type **Person** which have the id of the type **UUID**.

b) How to access the database using the Spring repositories?

There are different ways to access the database:

- Use one of the methods declared by the **JpaRepository**. The CRUD operations are implemented by default by the **JpaRepository** and it is not necessary to declare them again in the interface that extends it

```
@org.springframework.data.repository.NoRepositoryBean
public interface JpaRepository <T, ID> extends org.springframework.data.repository.
    java.util.List<T> findAll();
```

Figure 12 JpaRepository snippet

- Create methods based on the fields from the entity (e.g. findBy**Id**, findBy**Name**, etc.). In this case the name of the method is parsed and interpreted by the Spring framework in order to execute the corresponding query. Also, there is the possibility to create queries which are more complex with filters, join and so on (for more details please see the **JpaRepostiory** documentation)

```
    /**
     * Example: JPA generate Query by Field
     */
    List<Person> findByName(String name);
```

Figure 13 UserRepository interface

- Use custom defined queries – in the case of the custom defined queries the name of the method is not parsed; the purpose of the **@Param** annotation is to specify the names of the parameters which are used in the definition of the query

```
    /**
     * Example: Write Custom Query
     */
    @Query(value = "SELECT p " +
            "FROM Person p " +
            "WHERE p.name = :name " +
            "AND p.age >= 60 ")
    Optional<Person> findSeniorsByName(@Param("name") String name);
```

Figure 14 Custom Defined Queries

5. The PersonRepository retrieves a user entity object instantiated with values from the DB.

Good to know

a) What are the entities?

An entity represents a table from the relational database and each instance of the entity corresponds to a row from the database. An example of entity is shown in Figure 17.

```java
@Entity  15 usages  gobia
public class Person  implements Serializable{

    private static final long serialVersionUID = 1L;  no usages

    @Id
    @GeneratedValue
    @UuidGenerator
    @JdbcTypeCode(SqlTypes.UUID)
    private UUID id;

    @Column(name = "name", nullable = false)  3 usages
    private String name;

    @Column(name = "address", nullable = false)  3 usages
    private String address;

    @Column(name = "age", nullable = false)  3 usages
    private int age;


    public Person() {  gobia
    }

    public Person(String name, String address, int age) {  1 usage  gobia
        this.name = name;
        this.address = address;
        this.age = age;
    }
```

Figure 15 Person Entity

b) What are the main requirements for the creation of the entities?

- The entity class must be annotated with @Entity.
- The id and columns are mapped using @Id and @Column.
- The id can be automatically generated, as in this case using @GeneratedValue and @UuidGenerator for UUID primary keys.
- The class must have one public or protected no-argument constructor.

c) Which are the most common annotations used by the entities?

The most common annotations which are used in the mapping process are described below:

- @Entity – specifies the fact that the class which is annotated with this annotation is an entity
- @Table – specifies the table to which the entity is mapped
- @Id – the annotated field is an ID of the table
- @Column – the annotated fields are columns of the table from the database
- @GeneratedValue – marks the id field for automatic value generation
- @UuidGenerator – Hibernate-specific annotation for auto-generating UUIDs
- @JdbcTypeCode(SqlTypes.UUID) – ensures correct SQL type mapping for UUID columns
- @OneToOne – maps the one-to-one relationship between two tables
- @OneToMany – maps the one-to-many relationship between two tables
- @ManyToOne – maps the many-to-one relationship between two tables
- @ManyToMany – maps the many-to-many relationship between two tables

6. A Person entity is retrieved from the repository.
7. The entity is passed to the PersonService.
8. The PersonService converts the entity to a DTO using the static methods from the **PersonBuilder** class (**toPersonDTO** or **toPersonDetailsDTO**), which implements the Builder pattern for mapping between entities and DTOs.

Good to know

a) What are the DTOs?

The DTOs (Data Transfer Objects) are objects that carry data between processes and are exposed by the application to the UI or through an API.

b) What is the relation between the DTOs and the entities?

If the database changes then the mappings used by the entities must also change, but the objects (DTOs) might remain unchanged.

c) Why are the DTOs used?

The motivation for using the DTOs is represented by the fact that they reduce the cost of communication between the processes. The DTOs aggregate in one call data that might be transferred by several calls. Furthermore, DTO provide the option to selectively hide sensitive data, that otherwise would be exposed if using directly the entity object.

d) How to convert between entities and DTOs?

One common approach is to use a **builder class** with static conversion methods. For example, the PersonBuilder class provides toPersonDTO(), toPersonDetailsDTO(), and toEntity() methods to convert between the Person entity and its corresponding DTOs.

```
public class PersonBuilder {

    private PersonBuilder() {
    }

    public static PersonDTO toPersonDTO(Person person) {
        return new PersonDTO(person.getId(), person.getName(), person.getAge());
    }

    public static Person toEntity(PersonDetailsDTO personDetailsDTO) {
        return new Person(personDetailsDTO.getName(),
                personDetailsDTO.getAddress(),
                personDetailsDTO.getAge());
    }
}
```

Figure 16 PersonBuilder

9. A PersonDetailsDTO (or PersonDTO, depending on the request) is returned to the controller.
10. The Spring framework automatically uses the **Jackson converter** to serialize the DTO into a JSON response that is sent to the client.

```
{
"id": "45774962-e6f7-41f6-b940-72ef63fa1943",
"name": "Its me",
"age": 22
}
```

Figure 17 JSON with data from the DTO object

11. The Data is displayed in the browser

4. Validation and Error Handling

There are several options possible for validating the data received from a HTTP client, that aims to insert/update different resources managed by the server application. The most common one is to manually validate the data in code using different conditionals to check if certain fields are null, empty, etc. However, a more optimal approach is to use Annotation Based validation. For this there are several things you need to consider.

4.1. Annotation Based Validation

1. Add the necessary validation **dependencies** in pom.xml

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.2.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>6.0.2.Final</version>
</dependency>
```

Figure 18 Validation Dependencies

2. Mark the parameters that need to be validated by using the **@Valid annotation** in the Controller's methods:

```
@PostMapping()
public ResponseEntity<UUID> insertProsumer(@Valid @RequestBody PersonDetailsDTO personDTO) {
    UUID personID = personService.insert(personDTO);
    return new ResponseEntity<>(personID, HttpStatus.CREATED);
}
```

Figure 19 Apply Validation at method parameters level

Once the request gets in the controller, the request body object (personDTO) will automatically be validated according to the **validation rules**.

3. The **validation rules** are specified in the PersonDetailsDTO class through annotations:

```
@NotBlank(message = "name is required")  7 usages
private String name;
@NotBlank(message = "address is required")  7 usages
private String address;
@NotNull(message = "age is required")  7 usages
@AgeLimit(value = 18)
private Integer age;
```

Figure 20 Configure Validation Rules

There are predefined annotations, like @NotNull, however you can also define your own custom annotations that should be used as validator.

4. ***@AgeLimit*** is a custom constraint annotation that checks if the age field value is greater than or equal to the specified minimum (default 18). It uses the AgeValidator class, which implements ConstraintValidator<AgeLimit, Integer> to perform the validation.

The *AgeLimit* annotation is defined in com.example.demo.dtos.validators.annotation package.

```
17  @Documented
18  @Constraint(validatedBy = AgeValidator.class)
19  public @interface AgeLimit {
20      int value() default 18;                          // min age  gobia
21      String message() default "the age must be at least {value}";  gobia
22      Class<?>[] groups() default {};  gobia
23      Class<? extends Payload>[] payload() default {};  gobia
24  }
```

Figure 21 AgeLimit Annotation

Here you can define the error message (line 21), default minimum age (line 20), and, most importantly, the validator class that provides the actual validation logic (line 18).

```
10  public class AgeValidator implements ConstraintValidator<AgeLimit, Integer> {  2 usages  gobia
11      private int min;  2 usages
12  @   @Override public void initialize(AgeLimit ann) { this.min = ann.value(); }  gobia
13      @Override public boolean isValid(Integer age, ConstraintValidatorContext ctx) {  gobia
14          if (age == null) return true;              // let @NotNull enforce presence
15          return age >= min;
16      }
```

Figure 22 AgeLimit Validator

The AgeValidator class is defined in the com.example.demo.dtos.validators package. It implements the ConstraintValidator<AgeLimit, Integer> interface, specifying which annotation (@AgeLimit) it validates and the input type (Integer). During initialization, it loads the minimum age value from the annotation, and in the isValid() method it checks whether the provided age meets this limit. If the field is null, validation is delegated to @NotNull.

4.2. Error Handling

Whenever an exception occurs in the application — whether it's a validation error or a custom service exception — a centralized handler (RestExceptionHandler) captures it and returns a consistent JSON response.

The class is annotated with **@RestControllerAdvice** and is located in the com.example.demo.handlers package.

```
@RestControllerAdvice  gobia
public class RestExceptionHandler extends ResponseEntityExceptionHandler {

    private static final Logger log = LoggerFactory.getLogger(RestExceptionHandler.class);  1 usage

    @ExceptionHandler(ConstraintViolationException.class)  gobia
    public ResponseEntity<Object> handleConstraintViolationException(
            ConstraintViolationException ex, WebRequest request) {

        HttpStatus status = HttpStatus.BAD_REQUEST;
        Set<ConstraintViolation<?>> violations = ex.getConstraintViolations();
        List<String> details = violations.stream()  Stream<ConstraintViolation<...>>
                .map(v -> v.getPropertyPath() + ": " + v.getMessage())  Stream<String>
                .toList();
```

Figure 23 Application Level Exception Handler

The **RestExceptionHandler** extends **ResponseEntityExceptionHandler** and overrides several built-in methods such as **handleMethodArgumentNotValid**, **handleMissingServletRequestParameter**, and **handleHttpMessageNotReadable**. It also defines specific @ExceptionHandler methods for:

- *ConstraintViolationException* (validation errors from annotation constraints)
- *MethodArgumentTypeMismatchException* (wrong parameter types)
- *DataIntegrityViolationException* (database constraint violations)
- *CustomException* (application-defined errors like ResourceNotFoundException)
- a generic fallback for all unexpected Exception types.

Custom exceptions such as **ResourceNotFoundException** extend the base CustomException class in com.example.demo.handlers.exceptions.model, which includes fields for the HTTP status, resource name, and validation errors.

```
public class ResourceNotFoundException extends CustomException {  2 usages  gobia
    private static final String MESSAGE = "Resource not found!";  1 usage
    private static final HttpStatus httpStatus = HttpStatus.NOT_FOUND;  1 usage

    public ResourceNotFoundException(String resource) { super(MESSAGE,httpStatus, resource, new ArrayList<>()); }
}
```

Figure 24 Resource Not Found exception thrown

Validation errors from section 4.1 (for example, when a minor's age violates @AgeLimit) are caught by handleMethodArgumentNotValid, which returns a **400 Bad Request** with detailed messages inside an ExceptionHandlerResponseDTO.

5. Testing the Project

Use **Postman** to test the REST API endpoints defined in PersonController. You can import the provided collection file postman_collection.json, which contains preconfigured requests for all major operations:

- **GET** http://localhost:8080/people → retrieves all persons.
- **GET** http://localhost:8080/people/{uuid} → retrieves one person by ID.
- **POST** http://localhost:8080/people → inserts a new person.

1. Configure the database properties from the **src/main/resources/application.properties** file
2. Run the project as: Run Click on DemoApplication > Run 'DemoApplication'

   URL: localhost:8080/people/

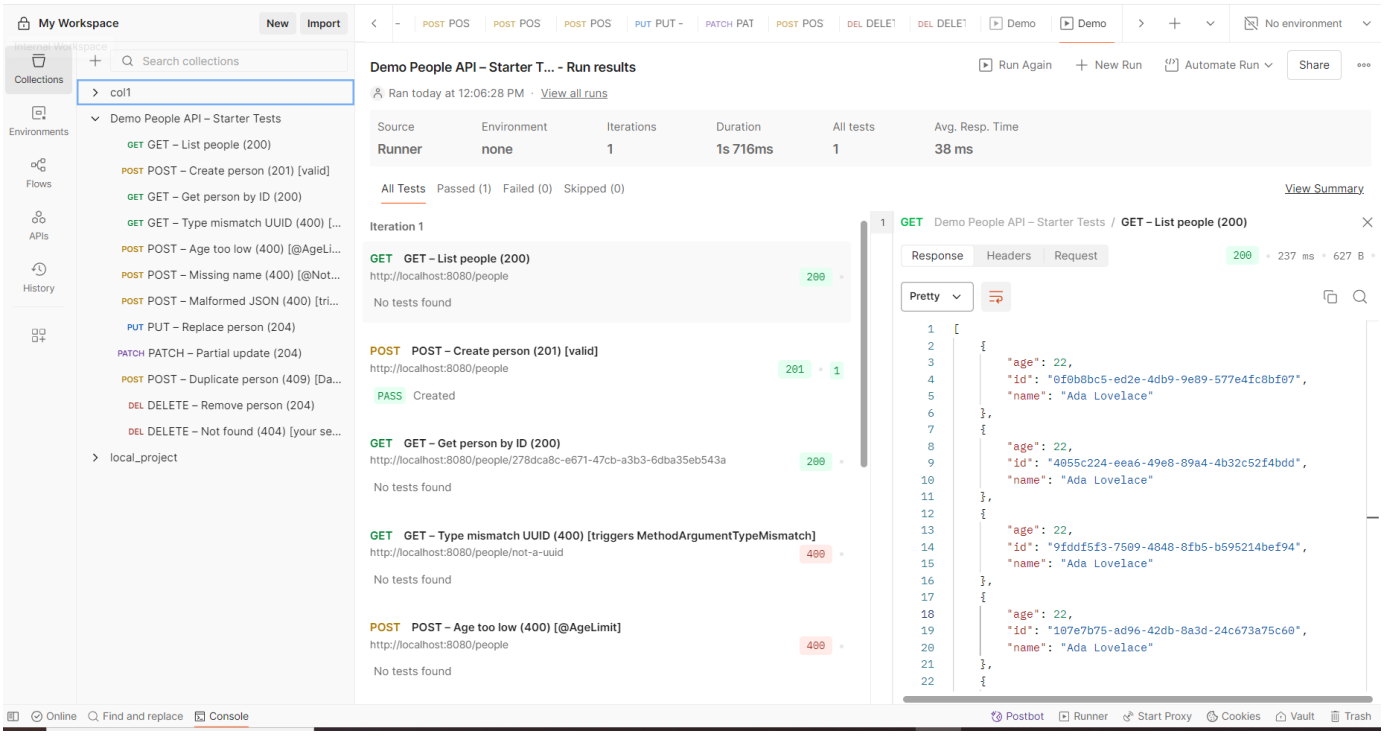

Figure 25 Example of using Postman tool

6. Deployment on Web Server

This project is packaged as a **JAR with embedded Tomcat**. No external Tomcat is required, only optional. For that, you should modify the packaging in pom.xml from JAR to WAR

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>ro.tuc</groupId>
    <artifactId>ds-2020</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>ds-2020</name>
    <description>Demo project for Spring Boot</description>
    <packaging>jar</packaging>
```

- Build the project (in IntelliJ: Maven-> Clean -> Install)
- Copy the WAR from **project/target** to **/apache-tomcat/webapps**
- Run Apache Tomcat: **/bin/startup.bat**

7. Reinforcement Learning

Answer the following questions:

- What is the Inversion of Control (IoC)?
- What is the Dependency Injection?
- Explain the @Autowired annotation.
- Explain the @Entity annotation.
- What is the Spring IoC container?
- What are the Spring beans?
- What is the purpose of @Service, @Repository, and @RestController?

8. References

[1] *Spring Framework Reference Documentation* — https://docs.spring.io/spring-framework/reference/
[2] *Spring Boot Reference Guide* — https://docs.spring.io/spring-boot/docs/current/reference/html/

[3] *Spring Guides – Building a RESTful Web Service* — https://spring.io/guides/gs/rest-service/
[4] *Spring Blog – What's new in Spring Boot* — https://spring.io/blog
[5] *Jakarta Persistence (JPA) Specifications* — https://jakarta.ee/learn/specification-guides/